JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 151–165 (1999)

Practice

Application of Cluster Algorithms for Batching of Proposed Software Changes

UWE KROHN and CORNELIA BOLDYREFF*
Centre for Software Maintenance, Department of Computer Science, University of Durham, Durham, DH1 3LE, U.K.

SUMMARY

This paper proposes the application of cluster algorithms for the identification of changes which may be batched together. The results of impact-analysis sessions are represented as binary data where each variable has two values indicating the presence or absence of an impact on a particular software component. These data are then used to produce a matrix containing the similarity or the dissimilarity of each pair of proposed changes which are to be clustered.

There are many clustering techniques for binary data. Most of the empirical investigations indicate that average-linkage and centroid-method clustering may be most useful in practice. Both clustering methods produced similar results in an example application. Proposed software changes that impacted a large number of the same components were merged early into common clusters, showing the maintainer which changes may be batched together.

KEY WORDS: impact analysis; software maintenance support; cluster analysis

1. INTRODUCTION

At every stage in the software life cycle, software has to be changed. A change to one software component is likely to result in consistencies in other components. Therefore it is necessary to identify all related components to rectify possible inconsistencies. This process is known as impact analysis (Turver and Munro, 1994; Queille *et al.*, 1994).

Impact analysis requires a knowledge of the dependencies between components of a software system. Components in impact analysis may refer to functions, data structures, or to complete documents such as requirement, specification or design documents. Dependencies may be calls between functions, uses of variables, or the relationship between a requirements and a design document (Fyson and Boldyreff, 1998).

Impact analysis begins with an initial change in some component followed by the identification of components, which could be impacted potentially in response to the initial change. This step can be automated with help of tools that trace chains of dependence in software systems. For each impact component, the maintainer must then decide how to implement the required change. This step is

*Correspondence to: Dr Cornelia Boldyreff, Centre for Software Maintenance, Department of Computer Science, University of Durham, Durham, DH1 3LE, U.K. Email: Cornelia.Boldyreff@durham.ac.uk

CCC 1040–550X/99/030151–15$17.50

Received 14 January 1998
Revised 4 February 1999

time consuming because it usually involves manual analysis of all the impacted components. Thus, it is more cost effective to implement changes which impact the same components together.

This paper proposes the application of cluster algorithms for the identification of changes which may be batched together. The results of impact-analysis sessions are represented as binary data where each variable has two values indicating the presence or absence of an impact on a particular software component. These data are then used to produce a matrix containing the similarity or the dissimilarity of each pair of proposed changes which are to be clustered.

There are many clustering techniques for binary data. Most of the empirical investigations indicate that average-linkage and centroid-method clustering may be most useful in practice. Both clustering methods produced similar results in an example application. Proposed changes that impacted a large number of identical components were merged early into common clusters, showing the maintainer which changes may be batched together.

2. IMPACT ANALYSIS

An impact analysis system has been developed as a part of the AMES project (Application Management Environments and Support (Boldyreff *et al.* 1995). The impact analysis system works on a representation of an application comprising a dependency model, a data model and a propagation model. These models describe an application in such a way that dependencies can be traced so as to identify all affected components (for a detailed description see Barros *et al.* (1995)).

The data model is a generic model of software systems. If defines object types and link types for a set of applications. Objects types in the data model represent generic components of software systems at some level of granularity, and link types represent the various dependencies that may exist between them.

The propagation model defines the modification types that can affect a particular object type. In addition, it defines propagation rules, which specify the way modification types are propagated along the link types connecting the object types. Given a set of propagation rules, the impact analysis system can identify those components which will (or may be) affected by a specified change. A rule can generate *real* impacts (i.e., modifications that *must* be performed), or *potential* impacts (i.e., modifications that *may* have to be performed). The maintainer has to check each *potential* impact to determine whether it is a *real* impact.

The dependency graph relates the data model and the propagation model to a particular application; it instantiates the generic concepts of these models with the actual objects and dependencies existing in the software system under consideration (see also Fyson and Boldyreff (1998). The dependency graph describes the dependencies that exist within the software system as a directed graph in which nodes represent the entities of the software system and links represent the dependencies between these entities.

A simplified example of a dependency graph for the kind of dependency information that can be extracted from C source code is given in Figure 1.

3. EXAMPLE OF IMPACT ANALYSIS

Impact analysis starts from a set of proposed modifications, which are propagated through the dependency graph in accordance with the previously defined propagation rules. The resulting

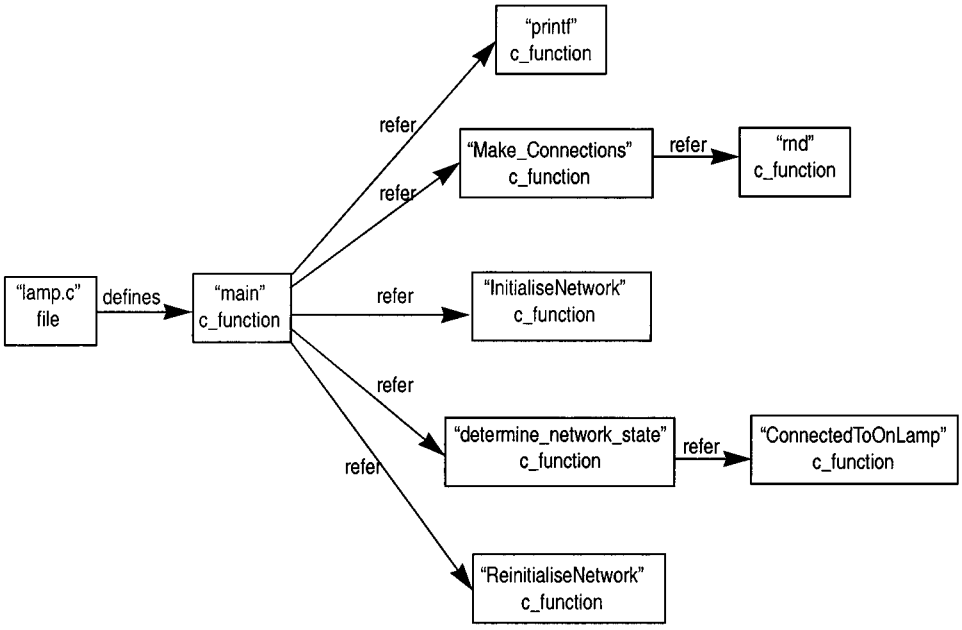

Figure 1. Dependency graph

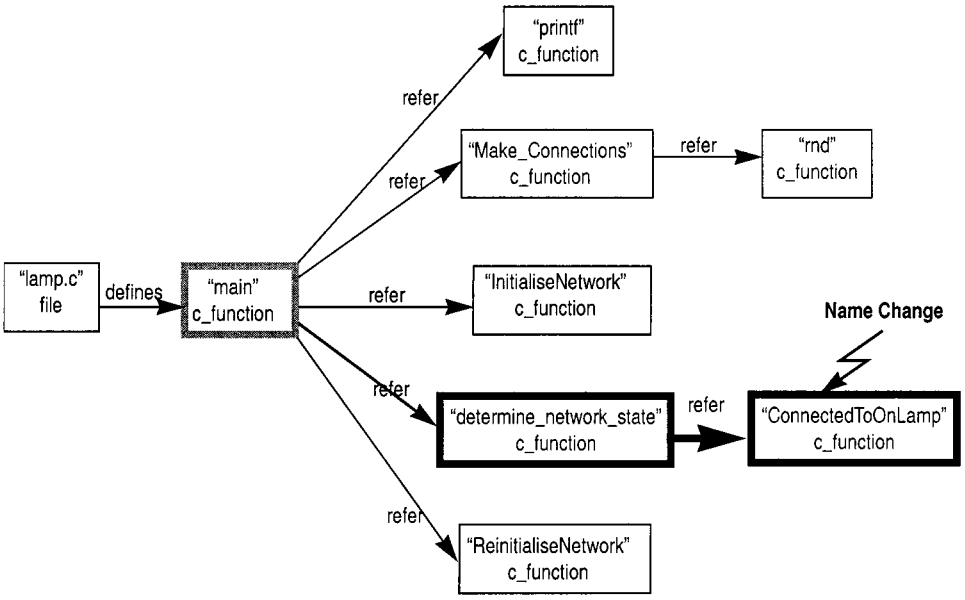

Figure 2. Propagation of name-change event on function `ConnectedToOnLamp`

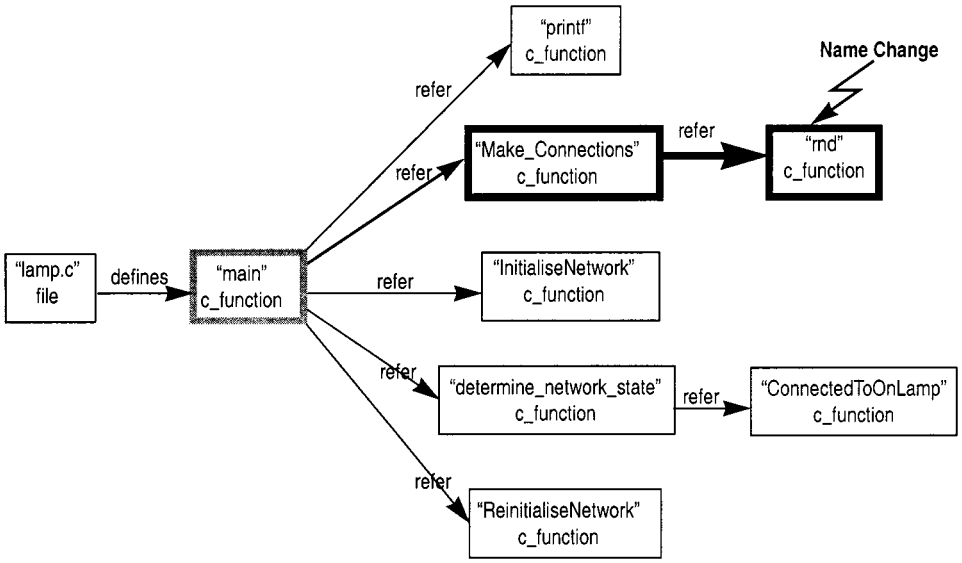

Figure 3. Propagation of name-change event on function `rnd`

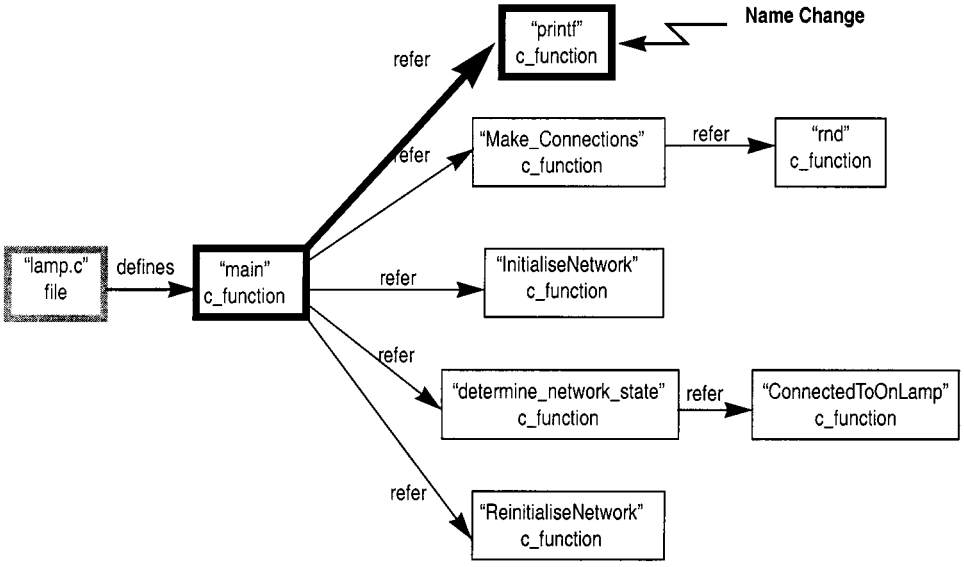

Figure 4. Propagation of name-change event on function `printf`

impacts are then recursively propagated so as to obtain the complete set of impacts for the initially proposed modifications.

Let us take the dependency graph in Figure 1 to illustrate four simple examples of impact analysis.

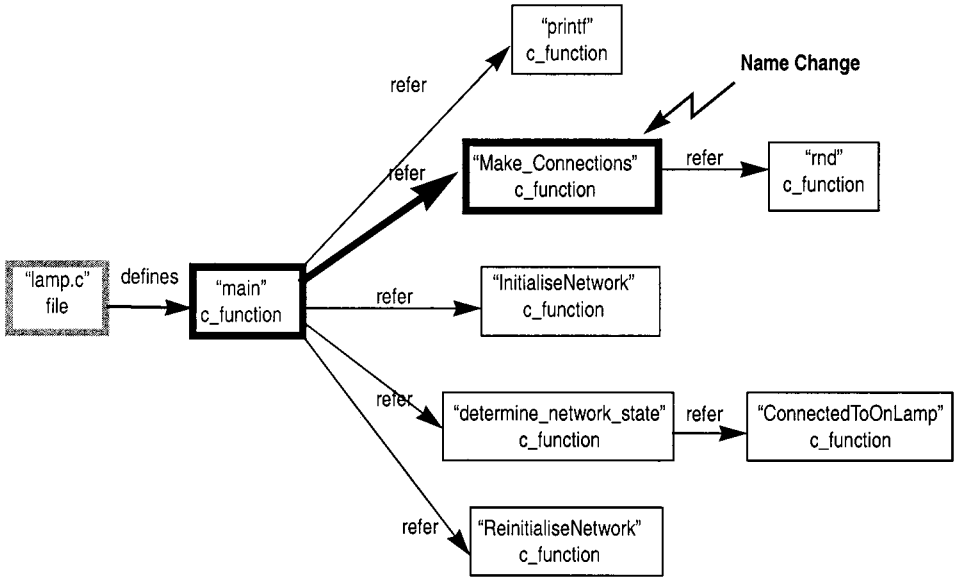

Figure 5. Propagation of name-change event on function `Make_Connections`

In each case the initial modification is to change the name of a function. Figure 2 illustrates the propagation of such a *name-change* event on the function `ConnectedToOnLamp`, Figure 3 on the function `rnd`, Figure 4 on function `printf` and Figure 5 on function `Make_Connections`. Bold-border boxes indicate real-impacted components, and grey-border boxes indicate potentially-impacted components.

A real impact of the initial name change is that all functions calling the renamed function must be modified in order to refer to the changed function name. The resulting impact in turn causes a potential impact on functions calling the impacted function and on the file defining the impacted functions.

The maintainer must check each potentially-impacted component to find out whether it is real impacted. For each real-impacted component, she must then implement the required changes. Obviously, it is more cost effective to implement changes which impact similar subgraphs together. The next section considers the application of cluster algorithms that allow the identification of changes which may be batched together.

4. CLUSTER ANALYSIS

Clustering can be considered as the grouping of similar objects using appropriate data from these objects. The input for cluster analysis is usually a matrix $\mathbf{X}$, giving the values of n variables for m objects:

$$\mathbf{X} = \begin{pmatrix} x_{11} \ldots x_{1n} \\ x_{m1} \ldots x_{mn} \end{pmatrix}$$

The aim of clustering is to find groups of similar objects, represented by the m rows of $\mathbf{X}$. Table 1

Table 1. Results of impact analyses represented as binary data

Name-change on	Connected ToOnLamp	determine_ network_state	Make Connections	main	printf	rnd
Connected ToOnLamp	1	1	0	0	0	0
Make_Connections	0	0	1	1	0	0
printf	0	0	0	1	1	0
rnd	0	0	1	0	0	1

summarizes the results of the impact analyses presented in Figure 2 to 5. The rows represent the four proposed changes and the columns represent the components impacted by these changes. The value 1 indicates the presence of a real impact, whereas the value 0 indicates absence of a real impact.

Table 1 contains binary data: each variable has two values only. The following matrix **X** represents these data in matrix form.

$$\mathbf{X} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

4.1. Similarity measures

For most clustering algorithms the data matrix **X** must be converted into a matrix containing the similarity or the dissimilarity of each pair of objects which are to be clustered. The term proximity is often used to refer to both types of measures. There are many different types of proximity measures, which weight data characteristics differently. Because the joining of clusters at each step depends on the proximity measure, different measures can result in different cluster solutions for the same clustering algorithm. The selection of a similarity or distance measure should be based on those characteristics in the data which are significant for the given application.

In our example we deal with binary variables; therefore, we have to consider similarity coefficients for binary variables. There are numerous similarity coefficients for binary data. The two coefficients most commonly used in practice are the *matching coefficient* and *Jaccard's coefficient*. The *matching coefficient* is defined as the number of variables on which two objects match, divided by the total number of variables. *Jaccard's coefficient* is the corresponding ratio when zero–zero matches are ignored.

The problem of whether to include the zero–zero matches is of special concern in the example of impact analysis discussed here, where the variables indicate either the presence or absence of impacts. It is unreasonable to consider the results of two proposed changes as very similar, simply because they produce no impacts on the same components. Hence, we choose Jaccard's coefficient rather than the matching coefficient. For our example matrix **X**, Jaccard's coefficient gives the

following similarity matrix:

$$\mathbf{S} = \begin{pmatrix} 0.0 & & \\ 0.0 & 0.3 & \\ 0.0 & 0.3 & 0.0 \end{pmatrix}$$

Once we have the similarity matrix, we can then proceed to form clusters of proposed changes that impact similar subgraphs. Next, a brief survey of clustering methods is given. The discussion is confined to hierarchical clustering methods, which produce more than one partition giving the maintainer the possibility to choose the most appropriate one.

4.2. Hierarchical clustering methods

Hierarchical clustering techniques can be split into agglomerative methods and divisive methods. Agglomerative methods proceed by successive fusions of the *m* objects into larger clusters, whereas divisive methods separate the *m* objects successively into smaller clusters. Both methods produce a two-dimensional tree known as a dendrogram, which illustrates the fusions or divisions made at each successive stage of the analysis.

Divisive methods construct the tree from the top–down. All objects start in a single cluster. This cluster is then split into two clusters, which are then split again. This process proceeds until all objects are in clusters of their own. Agglomerative methods, on the other hand, construct such a tree from the bottom–up. All objects start in clusters of one. Close clusters are then gradually merged until finally all objects are in a single cluster.

There are many methods for agglomerative clustering. The main difference between these methods consists of different definitions of distance (or similarity) between a single object and a cluster containing several objects or between two clusters. The most common agglomerative cluster methods are listed below:

- *Single-linkage clustering*—With single-linkage clustering the distance between two clusters is defined as that of the closest pair of objects.
- *Complete-linkage clustering*—Complete linkage is the opposite of single linkage in that the distance between two clusters is defined as that of the most distant pair of objects.
- *Group-average clustering*—Group-average clustering defines the distance between two clusters as the average of the distances between all pairs of objects.
- *Centroid method*—With this method the distance between two clusters is defined as the distance between the two cluster centroids.
- *Ward's method*—Within each cluster the sum of the squared Euclidean distances between the objects in the cluster and the cluster centroid is calculated. Two clusters are merged if the resulting cluster produces the smallest increase in the overall sum of the squared within-cluster distances.

4.3. Discussion of agglomerative clustering techniques

There are many empirical investigations that have compared a variety of hierarchical clustering methods on artificially generated data. Unfortunately, there is usually a rather large subjective component in the assessment of the results from any particular method (Manly, 1986, p. 104).

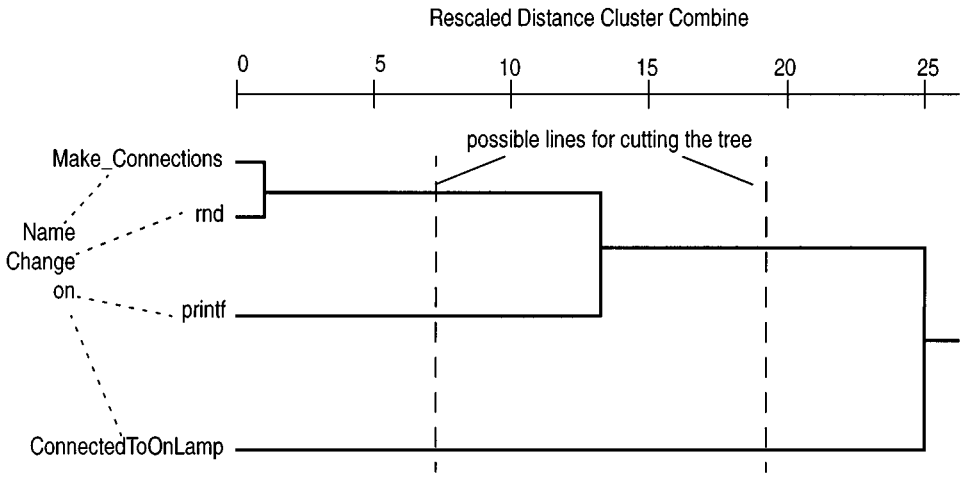

Figure 6. Dendrogram using average linkage (between groups)

The majority of the studies show clearly that no single method could be claimed superior for all types of data (Milligan, 1980; Everitt, 1993, p. 142). Rather, particular methods will only be best for particular types of data. Nevertheless, Everitt (1993) holds that some general recommendations can be made about techniques likely to be useful in practice: 'Amongst the hierarchical class . . . Ward's method and group average have been found to perform relatively well' (Everitt, 1993, p. 142). Cunningham and Ogilvie (1972), who compared seven hierarchical techniques, also found that average-linkage clustering performed best for the data sets considered.

For binary data Hands and Everitt (1987) discovered that 'Ward's method performed very well when the data contained approximately equally sized clusters, but poorly when the clusters were of different sizes. In the latter situation centroid clustering appeared to give the most satisfactory results' (Everitt, 1993, p. 72).

So, the results of the most investigations—in particular for binary data—indicate that average-linkage and centroid-method clustering may be the most useful in practice. Figures 6 and 7 give the average-linkage dendrogram and the centroid-method dendrogram for the Jaccard similarity matrix **S** of the example data.

5. INTERPRETATION OF RESULTS

The dendrograms show the clusters being combined and the proximities at which the clusters were joined. Dendrograms should be read from left to right. Vertical lines denote joined clusters. The position of a vertical line of the scale indicates the distance at which clusters were joined. The dendrograms, which were produced by SPSS, do not plot actual distances. SPSS rescales them to numbers between 0 and 25, such that the ratio of the distances between fusion steps is preserved.

From both dendrograms you can see that at the first step `Make_Connection` and `rnd` are combined into one cluster, at the second step this cluster is merged with `printf` and at the last step all four proposed changes are merged into one single cluster.

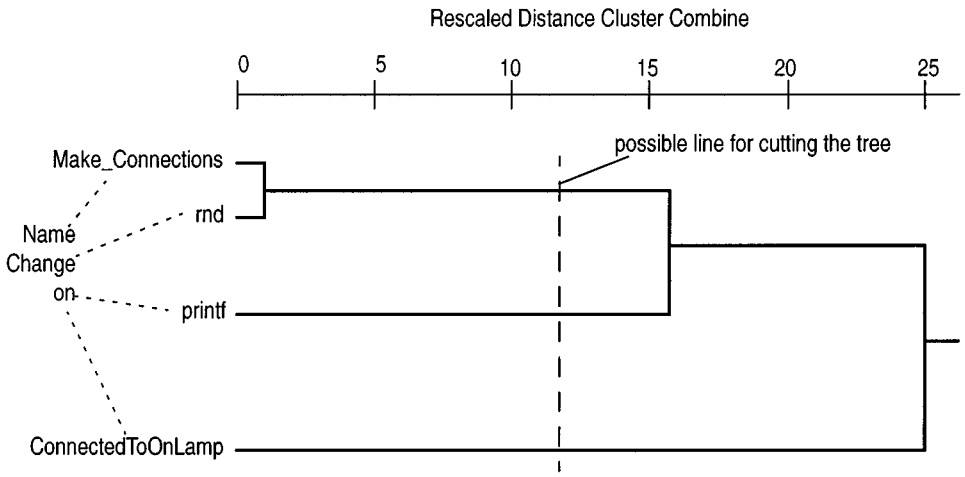

Figure 7. Dendrogram using the centroid method

The maintainer is certainly not interested in the complete dendrogram but only in one or two partitions, which she obtains by cutting the dendrogram at an appropriate level. One informal method for deciding at which level to cut the tree is to examine the differences between fusion levels in the dendrogram. Large changes in thefusion levels indicate an appropriate level for cutting the tree.

In the centroid-method dendrogram (see Figure 7) the greatest change in fusion level occurs between the merging of `Make_Connections` with `rnd` and the merging of the resulting cluster with `printf`. Thus, this dendrogram indicates a three-cluster partition with `Make_Connection` and `rnd` joined in one cluster and with `printf` and `ConnectedToOnLamp` in clusters of their own. This partition proposes that the maintainer should implement the changes on `Make_Connection` and `rnd` together.

In the average-linkage dendrogram (see Figure 6), there is only a slight difference between these two fusion levels; hence, additionally proposing a two-cluster partition with one cluster containing `Make_Connections`, `rnd` and `printf`; and another one containing only `ConnectedToOnLamp`.

6. CONCLUSIONS

This paper considered the application of cluster algorithms for the identification of changes which may be batched together. The results of impact-analysis sessions are represented by binary data. There are numerous similarity coefficients for binary data. The two coefficients most commonly used in practice are the *matching coefficient* and *Jaccard's coefficient*. The matching coefficient includes zero–zero matches, whereas Jaccard's coefficient ignores them. Jaccard's coefficient is more appropriate for computing the similarities between proposed changes, because it does not consider the results of two proposed changes as very similar, simply because they produce no impacts on the same components.

There are many methods for agglomerative clustering. For binary data, the results of most investigations indicate that average-linkage and centroid-method clustering may be most useful in practice. Both clustering methods produced similar results for the example application presented in this paper. Proposed changes that impacted a large number of identical components are merged early into common clusters, showing the maintainer which changes may be batched together. Cluster analysis is an appropriate method for facilitating the batching of proposed changes.

The dendrogram only shows the degree to which proposed changes are related to one another but not which components establish the relation among changes. The maintainer ultimately has to compare the impacted dependency graphs (provided by the impact analysis system) in order to determine the components which are impacted by more than one change. One approach, currently under investigation, to achieve this task is to merge the single impacted dependency graphs together into one graph and to visualize this graph structure. Fuller details of the models used and the source code from which they were derived are given in the following Appendix.

APPENDIX

A.1. The data model

A simplified example of a data model for the kind of dependency information that can be extracted from C source code is given in Section A.4.

For C source code, the following object types (`IAS_class`) and link types (`IAS_relation`) may be defined.

Each object type is uniquely identified by its name. Link types are specified by the type of origin and destination object, and in addition by direct and reverse relation names (Barros, Queille and Voidrot, 1996).

```
IAS_class "file"
IAS_class "c_function"
IAS_class "c_variable"

IAS_relation "file" "defines" "c_function" "is_defined_in"
IAS_relation "c_function" "refer" "c_function" "is_referred_by"
IAS_relation "file" "defines" "c_variable" "is_defined_in"
IAS_relation "c_function" "refer" "c_variable" "is_referred_by"
IAS_relation "c_variable" "refer" "c_function" "is_referred_by"
IAS_relation "c_variable" "refer" "c_variable" "is_referred_by"
```

A.2. The dependency graph

For a given C program, Figure 1 shows the objects and links, which may be defined on the basis of the object and link types defined here.

For a given C program, the following objects and links may be defined on the basis of the object and link types defined previously in the data model.

The dependency graph defines the actual objects within a given application and the relations between them. An object is specified by its type (defined in the data model) and by a unique name.

A link is characterized by the pair of objects it links and by its type (Barros, Queille and Voidrot, 1996).

```
IAS_object "lamp.c" file
IAS_object main c_function
IAS_object printf c_function
IAS_object rnd c_function
IAS_object Make_Connections c_function
IAS_object InitialiseNetwork c_function
IAS_object Unconnected c_function
IAS_object ConnectedToOnLamp c_function
IAS_object determine_network_state c_function
IAS_object ReinitialiseNetwork c_function

IAS_link "lamp.c" defines main;
IAS_link main refer printf;
IAS_link main refer Make_Connections;
IAS_link main refer InitialiseNetwork;
IAS_link main refer determine_network_state;
IAS_link main refer ReinitialiseNetwork;
IAS_link determine_network_state refer ConnectedToOnLamp;
IAS_link Make_Connections refer rnd;
```

A.3. The propagation model

This section gives some examples of propagation rules for C source code, using the sample data model given in Section A.1.

The propagation model defines event types, being categories of changes; object types to which the event types may be applied; and propagation rules, which indicate how events may propagate. A propagation rule has the following meaning: if an event of type e1 occurs on an object o1 of type c1, linked to another ob ject o2 of type c2 through a link of type r, then o2 is/may be affected by an event of type e2 (according to Barros, Queille and Voidrot (1996)). For C source code, the following event types and propagation rules may be defined on the basis of the object and link types defined in the example data model.

```
IAS_event_type "name_change" "file"
IAS_event_type "name_change" "c_function"
IAS_event_type "name_change" "c_variable"
IAS_event_type "definition_change" "c_variable"
IAS_event_type "definition_change_internal" "c_function"
IAS_event_type "semantic_change" "c_variable"
IAS_event_type "semantic_change_internal" "c_function"
IAS_event_type "semantic_change_global" "c_function"
IAS_event_type "text_change" "file"

IAS_propagation_rule "name_change" "file" "is_referred_by" "file" "text_change" "automatic"
IAS_propagation_rule "name_change" "c_function" "is_defined_in" "file" "text_change" "automatic"
IAS_propagation_rule "name_change" "c_function"  "is_referred_by" "c_function"
"definition_change_internal" "automatic"
```



```
IAS_propagation_rule "name_change" "c_function" "is_referred_by" "c_variable"
"definition_change" "automatic"
IAS_propagation_rule "name_change" "c_variable" "is_defined_in" "file" "text_change" "automatic"
IAS_propagation_rule "name_change" "c_variable"  "is_referred_by" "c_function"
"definition_change_internal" "automatic"
IAS_propagation_rule "name_change" "c_variable" "is_referred_by" "c_variable"
"definition_change" "automatic"
IAS_propagation_rule "definition_change" "c_variable" "is_defined_in" "file" "text_change" "auto-
matic"
IAS_propagation_rule "semantic_change" "c_variable" "is_referred_by" "c_function"
"semantic_change_internal" "automatic"
IAS_propagation_rule "semantic_change" "c_variable" "is_referred_by" "c_variable"
"semantic_change" "automatic"
```

A.4. The C source code

```
#include ⟨math.h⟩
#include ⟨stdio.h⟩
#include ⟨sys/types.h⟩
#include ⟨sys/time.h⟩

/*
Problem is described on page 39 and following in Alexander's
Notes on the Synthesis of Form (Alexander, 1964)

One hundred lamps
Each lamp is either on or off initially depending on whether C>.5
There is a 50-50 chance that if on, a lamp will remain on.
If a lamp is off at Tn and unconnected to a lamp which is on at Tn
   then it is off forever.
If a lamp is on at Tn then it is either off at Tn+1 or on at
   Tn+1 depending on whether C>.5
If a lamp is off at Tn and connected to a lamp that is on at Tn
   then it is (as above in the second case).
We need to keep track of the state of each lamp at Tn and Tn+1
i.e. start of time interval and end of time interval, and its
connections. NB no lamp is connected to itself.
There are various initial connection states:
(1) complete - all lamps connected to all other lamps
(2) null - no initial connections
(3) random connections
(4) partitioned - no of partitions
(5) mixture of (3) and (4)

*/

struct LAMP{
       int state_at_start_of_interval;
       int state_at_end_of_interval;
       int Connection[100];
    };
struct LAMP lamp[100];

int rnd()
{
  return((rand()/10000)%2);
```



```
}

Make_Connections(n)
int n;
/* there are 3 cases at present
     0 no connection at all
     1 all possible connections
     2 random

*/
 {
 int i, k;
 for (i=0; i<100; i++)
 for (k=0; k<100; k++)
 switch (n)
 {
  case 0: lamp[i].Connection[k] = 0; break;
  case 1: if (i!=k) lamp[i].Connection[k] = 1;
       else lamp[i].Connection[k] = 0; break;
  case 2: if (i!=k)
            if (rnd()) lamp[i].Connection[k] = 1;
            else lamp[i].Connection[k] = 0
       else lamp[i].Connection[k] = 0; break;
  }

 }

 InitializeNetwork()
 {
 /* There is a 50-50 chance that a lamp is on initially. */
 int i, r;
  for (i=0; i<100; i++)
  if (rnd())
     lamp[i].state_at_start_of_interval = 1;
  else lamp[i].state_at_start_of_interval = 0;
 }

 int Unconnected(l)
 struct LAMP l;
 {int i, NoOfConnections = 0;
  for (i=0; i<100; i++)
     NoOfConnections += (l.Connection[i]? 1:0);
  return(NoOfConnections == 0);
 }

 int ConnectedToOnLamp(l)
 struct LAMP l;
 {int i, on_connection = 0;
  for (i=0; i<100; i++)
     if ((l.Connection[i])&&(lamp[i].state_at_start_of_interval == 1))
      break;
  on_connection = (i<100? 1:0);
  return(on_connection);
 }

 int determine_network_state()
```



```
{
 int i, NoOfLampsOn = 0;
 for (i=0; i<100; i++)
  if ((!ConnectedToOnLamp(lamp[i])) && (lamp[i].state_at_start_of_interval ==0))

  lamp[i].state_at_and_of_interval = 0;
 else /* either it is on or connected to an on lamp or both */
  if ((lamp[i].state_at_start_of_interval == 1) || ConnectedToOnLamp(lamp[i]))
  if (rnd())
  {lamp[i].state_at_end_of_interval = 1; NoOfLampsOn++;}
  else lamp[i].state_at_end_of_interval = 0;
return(NoOfLampsOn);
}

ReinitializeNetwork()
{
 int i;
 for (i=0; i<100; i++)
  lamp[i].state_at_start_of_interval = lamp[i].state_at_end_of_interval;
}

main()

{
 int state_of_network, interval_counter = 0;
 Make_Connections(0);
 InitializeNetwork();
 do
 {state_of_network = determine_network_state();
  ReinitializeNetwork();
  interval_counter ++; printf("No of lamps on is %d\n", state_of_network);}
 while (state_of_network);
 printf("Death occured after %d interval(s).", interval_counter);
}
```

References

Alexander, C. (1964) *Notes on the Synthesis of Form*, Harvard University Press, Cambridge MA, 216 pp.

Barros, S., Bodhuin, Th., Escudie, A., Queille, J. P. and Voidrot, J. F. (1995) 'Supporting impact analysis: a semi-automated technique and associated tools', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 42–51.

Boldyreff, C., Burd, E. L., Hather, R. M., Mortimer, R. E., Munro, M. and Younger, E. J. (1995) 'The AMES approach to application understanding: a case study', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 182–191.

Cunningham, K. M. and Ogilvie, J. C. (1972) 'Evaluation of hierarchical grouping techniques: a preliminary study', *Computer Journal*, **15**, 209–213.

Everitt, B. S. (1993) *Cluster Analysis*, Edward Arnold, New York NY, 170 pp.

Fyson, J. M. and C. Boldyreff (1998) 'Using application understanding to support impact analysis', *Journal of Software Maintenance*, **10**(2), 93–110.

Hands, S. and Everitt, B. S. (1987) 'A Monte Carlo study of the recovery of cluster structure in binary data by hierarchical clustering techniques', *Multiv. Behav. Res.*, **22**, 235–243.

Manly, B. F. J. (1995) *Multivariate Statistical Methods: A Primer*, Chapman & Hall, London, 157 pp.

Milligan, G. W. (1980) 'An examination of the effect of six types of error perturbation on fifteen clustering algorithms', *Psychometrika*, **45**, 325–342.

Queille, J. P., Voidrot,J. F., Wilde, N. and Munro, M. (1994) 'The impact analysis task in software maintenance: a model and a case study', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 234–242.

Turver, R. J. and Munro, M. (1994) 'An early impact analysis technique for software maintenance', *Journal of Software Maintenance*, **6**(1), 35–52.

